**RC24662 (W0810-035) October 7, 2008**
**Mathematics**

# IBM Research Report

## A C++ Modelling Environment for Stochastic Programming

**Michal Kaut**
Molde University College
Molde, Norway

**Alan King**
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

**Tim Helge Hultberg**
European Organisation for the Exploitation of Meteorological Satellites
Darmstadt, Germany

# A C++ Modelling Environment for Stochastic Programming

**Michal Kaut** · **Alan King** · **Tim Helge Hultberg**

September 2008

**Abstract** Stochastic programming presents many challenges to modelling environments. One major challenge is to support the efficient processing of model artifacts as stochastic parameters change and scenarios are generated. In this paper we present an example that combines COIN-OR packages FLOPC++ and SMI to produce a C++ program, which generates and solves the extensive form of a stochastic program known as the deterministic equivalent. Quick updating of scenario data is enabled by writing a specialized subroutine that calculates the modified data. We also discuss what stochastic extensions would be needed to implement this capability using modelling abstractions.

**Keywords** stochastic programming · algebraic modelling languages and C++

## 1 Introduction

Modelling environments for stochastic programming must meet two opposing requirements: they must implement a modelling paradigm familiar to users of mathematical programming software and extend it for stochastic programming; and they must support sequential, frequent, and repeated processing of model artifacts during the solution of the stochastic program itself.

An example from financial investment illustrates the requirements. An investor wishes to solve a multiperiod asset allocation problem, where asset returns are driven by a log-normal factor model, and solve it using a simulation based method — for example, Stochastic Average Approximation. The algebraic model, with its inter-period wealth balance equations, portfolio bounds, transaction costs on trades, and so forth, is most naturally modelled using an algebraic modelling language. Monte Carlo simulation is the natural choice to generate

Michal Kaut
Molde University College, Molde, Norway; E-mail: michal.kaut@himolde.no

Alan King
IBM Watson Research Center, Yorktown Heights, New York, USA

Tim Helge Hultberg
European Organisation for the Exploitation of Meteorological Satellites, Darmstadt, Germany

the log-normal returns. The main point for us to consider is that the processing of these simulations into the optimization subproblems will require some kind of interaction with the algebraic model.

The processing of the simulations must be efficient in two respects. First, the size of the data communicated between modelling environment and optimization solvers must be small enough to allow for the distribution of the artifacts. For problems of practical size, this means that the modelling environment should not generate an entire deterministic equivalent in serial fashion — the processing of the simulations into the optimization subproblems must be able to be distributed. Second, the time it takes for the processing of each simulation must be very low. Stochastic programs of realistic size and detail will require the processing of millions of simulations, and sub-second processing times will be required. In this paper we try to address this point by showing, through an example, the abstractions that are required and must be made efficient.

Here is a brief overview of the literature on modelling software for stochastic programming. Birge et al. (1987) developed a standard for extending the MPS format to stochastic programs, which has since become known as the Stochastic MPS (SMPS) format, and some extensions were developed in Gassmann and Schweizer (1996). SMPS was intended to promote communication of problem instances between stochastic programming algorithm developers, and there is now a reasonable collection of problem instances available at the stochastic-programming community home page, `http://www.stoprog.org`. The SMPS format was never intended to support modelling facilities, however. Gassmann and Ireland (1995) demonstrated how to generate an explicit representation of a stochastic program using the subset capabilities of the modelling language AMPL. Condevaux-Lanloy et al. (2002) discussed a technique for separating scenario generation from the modelling language, by using the modelling language GAMS as a matrix generation service. An interesting proposal for recursive model definition, as might be applied to Markov processes, appeared in Buchanan et al. (2001). Valente et al. (2008) describes a way to extend AMPL to handle stochastic programming problems; this has later become part of the SPInE integrated environment, (Valente et al. 2005), which includes also a stochastic solver called FortMP. Another extension of AMPL can be found in Fourer and Lopes (2008). MPL (Kristjansson) has support for stochastic programming. Finally, the SLP-IOR model management system, see (Kall and Mayer 2005), maintains a collection of stochastic programs and solvers in an integrated environment for two-stage stochastic programs.

This paper documents a C++ modelling environment for stochastic programming based on two projects, FLOPC++ and SMI, in the COIN-OR open source repository. Rather than describing the entire stochastic program in the modelling environment, it follows the basic pattern of the SMPS standard in separating the definition of the core data from the definition of the stochastics data. Our approach builds on the fact that the FLOPC++ modeling facilities are provided by C++ classes, so that the entire process of scenario generation can take place within the C++ runtime. It is our hope that this will lead to a highly efficient process of scenario generation within an algebraic modeling environment, however we do not examine the runtime efficiency in this paper.

The paper is in three parts. The first part presents the motivation for our work, together with an investment example we are going to model later in the paper. The second part describes the two projects our solution is based on, namely SMI for efficient processing of stochastic programming data into optimization subproblems and FLOPC++ for processing algebraic modelling statements inside C++. The third part develops our proposal for the stochastic programming modelling environment, demonstrated on the investment example. We finish with a discussion of the C++ approach and its possible generalizations.

## 2 Motivation

A simple example from the literature on stochastic programming for financial applications illustrates the requirements of a modelling system for stochastic programming. It is based on an the “Financial Planning and Control” example from (Birge and Louveaux 1997, pp. 20–28). Applications of this type are canonical in financial portfolio management.

An investor wishes to use a trading program to maximize the expected value of a function of terminal wealth at a given future date. The program starts with a specified initial wealth $w_0$, and specifies proportions of that wealth $x_0^j$ to be allocated initially to a given list of assets $j = 1, \dots, J$.

$$\sum_{j=1}^{J} x_0^j = w_0 \tag{1}$$

At each trading date $t = 1, \dots, T$, the program observes the realized vector of returns $r_t$ at date $t$ and calculates the new wealth:

$$\sum_{j=1}^{J} r_t^j x_{t-1}^j = w_t. \tag{2}$$

Then it specifies a new allocation $x_t$:

$$\sum_{j=1}^{J} x_t^j = w_t. \tag{3}$$

At the final period $T$, the program calculates the function of terminal wealth. The investor has a target wealth in mind $v_T$ and wishes to optimize a function of the difference $w_T - v_T$. Here we use a function that assigns a linear weight of 1.1 to positive outcomes and a penalty weight of 1.3 to negative outcomes:

$$f(w_T - v_T) = \begin{cases} 1.1\,(w_T - v_T) & \text{if } w_T \geq v_T \\ -1.3\,(v_T - w_T) & \text{if } w_T \leq v_T \end{cases} \tag{4}$$

The program maximizes its expected value over the distribution of the uncertain returns:

$$\max \mathbf{E} f(w_T - v_T) \tag{5}$$

The domain of maximization of (5) includes all allocations $x_t$ satisfying (1) for $t = 0$, and (2)–(3) for all dates $t = 1, \dots, T$ that are non-negative and *non-anticipative*. Non-anticipativity requirements distinguish stochastic programs from their deterministic counterparts. In this example, when the program optimizes an allocation $x_t$ it takes into account the impact of the allocation on the *conditional* expected value of $f(w_T)$ *given* the past observations $(r_1, \dots, r_t)$ and the current wealth state $w_t$. The allocation $x_t$ thus depends only on the past observations.

When distributions are discretely distributed it is convenient to describe the non-anticipativity structures using the device of a *scenario tree*. Sample paths of length $t$ correspond one-to-one with nodes in the collection $\mathcal{N}_t$ at level $t$ in the scenario tree. Nodes in $\mathcal{N}_t$ transition to the nodes in $\mathcal{N}_{t+1}$ that share a common history of length $t$. The set of child nodes to which it is possible to transition from a given node $n \in \mathcal{N}_t$ is denoted by the set $n^+ \subset \mathcal{N}_{t+1}$. The parent of a node $n \in \mathcal{N}_t$ is denoted $n^- \in \mathcal{N}_{t-1}$. Complete paths from root to leaf are called scenarios. Probability $p_n$ is assigned to nodes $n$ in the natural way, by multiplying the conditional transition probabilities along the sample path connecting the node to the root.

Scenario trees are a useful way to illustrate the challenges of modelling stochastic optimization problems. The trading problem (1)–(5) can easily be reformulated using scenario tree semantics:

$$
\begin{array}{lll}
\max & \sum_{n\in\mathcal{N}_T} p_n f(w_n - v_T) & \\
 & \sum_{j=1}^{J} x_0^j & = w_0 \\
\text{subject to} & \sum_{j=1}^{J} r_n^j x_{n^-}^j & = \sum_{j=1}^{J} x_n^j \;\; (n \in \mathcal{N}_t,\, t = 1,\ldots,T-1) \\
 & \sum_{j=1}^{J} r_n^j x_{n^-}^j & = w_n \qquad (n \in \mathcal{N}_T),
\end{array}
\tag{6}
$$

where for clarity in subsequent discussions, we have eliminated the redundant wealth variables at the intermediate dates. This formulation is called the *deterministic equivalent*. The allocations $x_n$ are explicitly non-anticipative because they are indexed over nodes in the scenario tree, hence this is an equivalent formulation. It is deterministic because it explicitly describes all possible sample paths in the probability space.

## 3 Modelling Challenges

Let us now use this example to illustrate the main issues concerning modelling for stochastic programs.

It is natural to use an algebraic modelling language (AML) to describe the constraint system (1)–(3). Algebraic modelling languages can parse these algebraic statements and declare place-holders for index sets, data, variables, and constraints. Many modelling languages implement library interfaces to databases that enable index sets and data to be passed automatically from tables generated by SQL statements.

To model the distribution of returns using an algebraic modelling language is also quite natural for an AML. Autoregressive processes, factor models, and the like, can be modeled using algebraic operations on the problem data and simulated from standard libraries for random number generation. Inputs for these simulations can be drawn from databases containing the various parameters and historical information.

The main difficulty for the AML lies in combining the constraint system with the distribution data to produce the artifacts that are to be passed to a solver. In the case of a linear programming solver, the matrix, right-hand side, and objective coefficients are assembled by the AML and transported across an interface to the solver. Then solution information and solution arrays are collected from the solver and marshaled into the AML-managed data structures. Because the number of nodes in the scenario tree is exponential in the time index and the base of this exponentiation is on the order of the number of states in the transition distributions, for many problems of realistic size the deterministic equivalent stochastic program (6) is practically impossible to handle in this manner. So, while it is nominally possible to generate a deterministic equivalent in an AML environment, this capability is not likely to be useful in many practical applications.

Indeed, it has long been accepted that this “curse of dimensionality” for stochastic problems implies that effective solvers will rely on parallel decomposition and simulation algorithms; see Birge (1997). The challenge for the modelling environment is then how to interact with solvers that require artifacts for the parallel generation of decomposition subproblems and simulations.

## 4 COIN-OR libraries

The **Co**mputational **In**frastructure for **O**perations **R**esearch (COIN-OR, see Lougee-Heimer (2003)) is a collection of open-source projects developing software for the operations research (OR) community. According to the COIN-OR web page[1], the goal of the project is "*to create for mathematical software what the open literature is for mathematical theory*". Most of the projects are released under the *Common Public License* (CPL), an open-source software license published by IBM and approved by the Open Source Initiative and Free Software Foundation (but not compatible with GPLv2). The COIN-OR project is managed by a non-profit educational and scientific foundation *The COIN-OR Foundation, Inc.*

The proposed stochastic-programming tool is built upon the following two COIN-OR projects: FLOPC++ for a modelling language and SMI for a stochastic modelling interface. These are introduced in the following sections. In addition, we will be needing some generic COIN-OR constructs. The most important of these is OSI, which stands for *open solver interface*, COIN-OR API for interacting with callable LP and (M)IP solver libraries. The rest will be introduced when needed.

### 4.1 FLOPC++

The ***F**ormulation of **L**inear **O**ptimization **P**roblems in **C**++* (FLOPC++, see Hultberg (2007)) is an algebraic modelling language implemented as a C++ library. In other words, it makes it possible to specify a linear model in C++ in a way resembling the traditional modelling languages like AMPL or GAMS—with the additional advantage of having the full strength of the C++ language for manipulating (solving, updating, resolving) the model once it has been created. Or, as the FLOPC++ web page[2] puts it, "*FLOPC++ can be used as a substitute for traditional modelling languages, when modelling linear optimization problems, but its principal strength lies in the fact that the modelling facilities are combined with a powerful general purpose programming language. This combination is essential for implementing efficient algorithms (using linear optimization for subproblems), integrating optimization models in user applications, etc.*"

### 4.2 SMI

SMI[3] stands for **S**tochastic **M**odeling **I**nterface and is meant to be an interface for stochastic-programming models. The list of currently implemented features includes

- a scenario tree structure for multiperiod stochastic data
- an implementation of a Stochastic MPS (SMPS) reader
- interfaces for generating scenario trees from paths and from discrete random variables
- generating an OSI object with the deterministic equivalent problem
- parsing the solutions by stage and scenario.

The most important missing feature is a stochastic solver, which is currently being worked upon. This means that, for now, the only way to solve the stochastic program represented by an SMI object is to create an OSI object with its deterministic equivalent and use some of the OSI solvers to solve it.

[1] See `http://www.coin-or.org/`
[2] See `https://projects.coin-or.org/FLOPC++`
[3] See `https://projects.coin-or.org/Smi`

The structure of an SMI object is closely related to the SMPS format: we start by creating a core model and then associate its variables and constraints with stages. The scenario tree is then created in an SMPS-like fashion by adding scenarios one by one, specifying their parent scenario, the stage they branch from the parent scenario and the data in which they differ from it. Note that SMI also provides a way of generating scenarios using discrete distributions, but we will not need this feature in this paper.

## 5 Modelling the investment example

In this section, we demonstrate the possibilities of combining the FLOPC++ and SMI packages on the investment examples presented in Section 2. We will only present the relevant parts of the code. The full source code can be obtained as file `investment.cpp` in the examples folder of the SMI project.

### 5.1 The StageNodeBase class

We first discuss a generic base class `StageNodeBase` for stage-nodes. This is a problem-independent base class that provides basic functions for accessing stage-dependent information of the core model. It contains pointers to its predecessor and successor nodes, and provides “meta-objects” to retain links to all variables and all constraints that will belong to a node. We also include a member to express the objective function at the node, modelled as a FLOPC++ `MP_expression` object. The class declaration is presented in Figure 1.

```
class StageNodeBase {
public:
  StageNodeBase *ptParent;     // pointer to parent node
  StageNodeBase *ptChild;      // pointers to children of this node
  MP_expression objFunction;   // objective function at this node

  vector<VariableRef *>   all_variables;     // references to all variables
  vector<MP_constraint *> all_constraints;   // references to all constraints

  StageNodeBase(StageNodeBase *ptPred)
  : ptParent(ptPred), ptChild(NULL)
  {
    if (ptParent != NULL)
      ptParent->ptChild = this;   // Register with the parent
  }

protected:
  virtual void make_obj_function_() {
    if (ptChild)
      ptChild->make_obj_function_();
    objFunction = ptChild->objFunction;   // Add node objective to this line
  }
};
```

**Fig. 1:** The `StageNodeBase` generic base class.

The `StageNodeBase` constructor assumes that each node’s parent is created before the node itself. The node’s constructor gets a pointer to the parent node and then registers itself as the child of its parent. (Note that all the class members are declared as `public` only for the sake of brevity. Normally, one would most likely make them `private` and then write separate get/set methods. This would, however, unnecessarily clutter the presentation.)

```
void StageNodeBase::make_obj_function_() {
  if (ptChild) {
    ptChild->make_obj_function_();        //recursive call down chain to leaf.
    objFunction = ptChild->objFunction;   //add any objective to this expression.
  }
}
```

**Fig. 2:** The `make_obj_function_()` method.

The `StageNodeBase` class includes a `make_obj_function_()` method that creates the objective function and puts it into the `objFunction` member (see Figure 2). The method is intended only to be called from the root node, so it is declared `protected`. Only the root node will be given a public interface. The `make_obj_function_()` method recurses down to the leaf node (the last stage). Each node assigns its `objFunction` object to the child objective and adds its own objective expression, if any, to the line indicated by the comment. If a node needs to add something to the objective, then this method will have to be over-ridden.

## 5.2 Stage Node classes for the investment example

In the investment problem there are only three types of submodels: one for the root of the tree, one for the leaf nodes and one for all the intermediate nodes of the tree. Each node's model has, obviously, different data, but the model structure is always one of the three types. We therefore write three C++ classes for the three models, using the FLOPC++ modelling constructs.

The `StageNode` class, presented in Figure 3, includes the model-dependent entities that are common for all the nodes of the tree, modelled as FLOPC++ objects. In our example, this means the set `ASSETS` and variables `x` and `wealth`. In addition, we define an index `a`, for indexing over `ASSETS`. (These are not needed in the leaf nodes, as we shall see, but declaring them for all nodes does no harm.) In the constructor, the set `ASSETS` is defined to have `nmbAssets` members and the declaration `x(ASSETS)` declares the variable `x` to be indexed over that set. The equation `wealth_defn` in every period allocates the variable `wealth` among the investment variables.

The constructor of the `StageNode` class displays one of the basic patterns for integrating FLOPC++ with SMI. The class declares variables (using `MP_variable`) and constraints (using `MP_constraint`), and then adds them to their respective containers `all_variables` and `all_constraints`. This pattern is repeated in each of the derived classes. As we shall see, it is convenient to have these containers when integrating with the SMI procedures.

The member `get_wealth()` demonstrates part of the pattern for retrieving values from a solution. In this case, the method accesses the value of the `wealth` variable from a solution vector. The position of the wealth variable `wealth` in the vector is determined from the method `wealth().getColumn()`.

The pattern for changing scenario data is displayed in the `load_modified_matrix()` method (see Figure 4). This method updates the core matrix with the changed data values for

```
class StageNode: public StageNodeBase {
public:
  MP_set ASSETS;                 // set of assets
  MP_index a;                    // index used in formulas
  MP_variable x;                 // the "buy" variable, defined on ASSETS
  MP_variable wealth;            // the wealth at each period
  MP_constraint wealth_defn;     // the equation defining wealth

  StageNode(StageNode *ptPred, const int nmbAssets)
  : StageNodeBase(ptPred), ASSETS(nmbAssets), x(ASSETS)
  {
    wealth_defn = sum(ASSETS(a), x(a)) == wealth();

    all_variables.push_back(new VariableRef(wealth()));
    for (int a = 0; a < nmbAssets; a++) {
      all_variables.push_back(new VariableRef(x(a)));
    }
    all_constraints.push_back(&wealth_defn);
  }

  StageNode *get_parent() {
    return (StageNode *) ptParent;
  }

  virtual double get_wealth(const double *variableValues, const int nmbVars) {
    return variableValues[wealth().getColumn()];
  }

  MP_constraint *balance_constraint;
  virtual void load_modified_matrix(CoinPackedMatrix &ADiff, double *retData);

};
```

**Fig. 3:** The `StageNode` class.

```
void load_modified_matrix(CoinPackedMatrix &ADiff, double *retData) {
  int i = balance_constraint->row_number();
  for (int a = 0; a < ASSETS.size(); a++) {
    int j = this->get_parent()->x(a).getColumn();
    ADiff.modifyCoefficient(i, j, retData[a]);
  }
}
```

**Fig. 4:** The `load_modified_matrix` method.

each scenario. In the case of our example, the data values that change with the scenario are the returns. The row that is to be updated is pointed to by the `balance_constraint` object. The indices of the matrix elements to be updated correspond to the indices of the asset holdings $x$ from the previous period – this is why the method accesses the `x(a)` members from the *parent* node. The method `get_parent()` simply retrieves the parent pointer and casts it to a `StageNode` pointer.

## 5.3 Root, MidStage, and LastStage Node classes

From the `StageNode` class, we derive the three node-type classes that will be actually used in the model: `RootNode` (Figure 5), the `MidStageNode` class (Figure 6), and finally, the `LastStageNode` class (Figure 7).

```
class RootNode: public StageNode {
public:
  MP_constraint initialBudget;   // initial budget constraint

  RootNode(const int nmbAssets, const double initWealth
  : StageNode(NULL, nmbAssets)
  {
    initialBudget() = wealth() == initWealth;

    all_constraints.push_back(&initialBudget);
    balance_constraint = NULL;
  }

  // Public interface to the protected make_obj function_()
  void make_objective_function() {
    make_obj_function_();
  }
};
```

**Fig. 5:** The `RootNode` class adds an initial-wealth constraint, and the public interface for `make_obj_function_`.

```
class MidStageNode: public StageNode {
public:
  MP_constraint cashFlowBalance;   // cash-flow balance constraint
  MP_data Return;                  // returns of the assets at this node

  MidStageNode(StageNode *ptPred, double *ptRetVect)
  : StageNode(ptPred, ptPred->ASSETS.size()), Return(ptRetVect, ASSETS)
  {
    cashFlowBalance = sum(ASSETS(a), get_parent()->x(a) * Return(a))
                      == wealth();

    all_constraints.push_back(&cashFlowBalance);
    balance_constraint = &cashFlowBalance;
  }
};
```

**Fig. 6:** The `MidStageNode` class adds the cash-flow-balance constraint that links the node's wealth to its parent's investments.

The `RootNode` defines the initial-wealth constraint, which ensures that positions in the investments add up to the starting wealth. The `MP_constraint` object created by this definition is then added to the list of constraints that belong to the `RootNode`. The initial-wealth constraint pointer is stored in the parent class's `balance_constraint` for use in the `load_modified_matrix()` method.

```
class LastStageNode: public MidStageNode {
public:
  MP_variable w;           // shortage variable
  MP_variable y;           // surplus variable
  MP_constraint penalty;   // equation defining the surplus and shortage

  LastStageNode(StageNode *ptPred, double *ptRetVect, const double capTarget)
  : MidStageNode(ptPred, ptRetVect)
  {
    penalty = wealth() + w() - y() == capTarget;

    all_variables.push_back(new VariableRef(w()));
    all_variables.push_back(new VariableRef(y()));
    all_constraints.push_back(&penalty);
  }
protected:
  // version of make_obj_function_() for the leaves - no recursion
  void make_obj_function_() {
    objFunction = 1.3 * w() - 1.1 * y();
  }
};
```

**Fig. 7:** The `LastStageNode` class adds the objective (4), modelled using a shortage and surplus variable.

The `MidStageNode` class contains only one constraint which is the cash-flow balance constraint. Finally, the `LastStageNode` class contains the variables and constraints that model a piecewise linear penalty function around the capital target. The `LastStageNode` class does not really make use of its `x` variables that it inherits from `StageNode`, but including these superfluous variables does no harm. If we want to avoid this, we would have added a new class for the non-leaf nodes, derived from the `StageNode` class, and derived the `RootNode` and `LastStageNode` classes from it.

The code for `StageNode` and its derived classes displays the basic patterns that will enable us to integrate the FLOPC++ constructions with SMI. Each node class defines the constraints and variables that belong to its particular stage, and adds them to the containers `all_constraints` or `all_variables`, respectively. These containers allow us to identify the variables and constraints with their stage membership. The `get_wealth()` method shows how each class can use the FLOPC++ interfaces to locate solution values. Finally, each class declares a“modify” member that describes how incoming stochastic data modifies the core linear program data and allows us to generate the scenarios, as we shall see.

## 5.4 Scenario-tree structure

To further support the creation of stochastic programs, we develop classes for scenario-tree structures. The base class is shown in Figure 8.

For our example, we use a derived class `BinTreeStruct` for binary trees (see Figure 9). The class `BinTreeStruct` implements the logic for getting scenario information necessary for SMI to construct scenarios. Upon each call to `get_next_scenario()` it returns a vector of node numbers, a unique scenario number, an ancestor scenario, a branching stage, and a probability. Thus, this class encapsulates the logic of the branching pattern of the scenario tree. The vector of node numbers, in our implementation, are offsets into the `retData` array,

```
class ScenTreeStruct {
public:
  int nmbNodes;    // nodes are 0..nmbNodes-1, where 0 is root
  int firstLeaf;   // nodes firstLeaf..nmbNodes-1 are leaves

  ScenTreeStruct(const int nNodes, const int firstL)
  : nmbNodes(nNodes), firstLeaf(firstL) {}

  virtual int get_parent(int n) const = 0;
  virtual int get_nmb_stages() const = 0;
  virtual void get_core_scenario(int *scenNodeNmb) const = 0;
  virtual void get_next_scenario(int *scenNodeNmb, int *scen, int* parentScen,
                                 int *branchStage, double *prob) const = 0;
};
```

**Fig. 8:** The `ScenTreeStruct` virtual base class for scenario trees.

```
class BinTreeStruct : public ScenTreeStruct {
private:
  int nextLeaf;
public:
  // Binary tree has 2^T-1 nodes and the first leaf is 2^(T-1)-1.
  BinTreeStruct(const int T)
  : ScenTreeStruct((int) pow(2.0, T) - 1, (int) pow(2.0, T-1) - 1), nmbStages(T)
  {
    nextLeaf = this->firstLeaf;
  }

  int get_parent(int n) const { return (n-1) / 2; }  // get parent(0) = 0

  void get_core_scenario(int *scenNodeNmb) {
    int n = this->firstLeaf;
    for (int t = nmbStages; t > 0; t--) {
      scenNodeNmb[t-1] = n;
      n = this->get_parent(n);
    }
  }

  void get_next_scenario(int *scenNodeNmb, int *scen, int* parentScen,
                         int *branchStage, double *prob) {
    if (nextLeaf == nmbNodes)
      return NULL;
    int n = nextLeaf;
    int t = nmbStages-1;
    while (n != scenNodeNmb[t]) {  // add nodes not already in list
      scenNodeNmb[t] = n;
      n = this->get_parent(n);
      t--;
    }
    *scen = nextLeaf - this->firstLeaf;           // scenario index
    *parentScen = (*scen == 0 ? 0 : *scen - 1);   // parent scenario
    *branchStage = (*scen == 0 ? 1 : t+1);        // branching scenario
    *prob = 1.0 / this->getNmbScen();             // equiprobable scen.
    nextLeaf++;
  }
};
```

**Fig. 9:** The `BinTreeStruct` class.

which is declared in the main program. For better encapsulation this could be placed inside the binary tree object, if desired.

## 5.5 Putting it all together

Now, we are ready to construct the whole model. We start by creating the node models for the core, i.e. the first scenario of the tree. The code for this part is presented in Figure 10. Note that we present only the relevant parts of the code, skipping for example some "natural" definitions, like saying that `i` and `j` are integers.

```
BinTreeStruct binTree(nmbStages);                       // scenario-tree object
double retData[nmbScen][nmbAssets];                     // scenario data
MP_model &mpCoreModel = MP_model::getDefaultModel(); // FlopC++ model

int * scenNodeNmb = new int[nmbStages];        // nodes (indices) in a scenario
binTree.get_core_scenario(scenNodeNmb);

vector<StageNode *> coreNodes(nmbStages);   // node-objects for the core

coreNodes[0] = new RootNode(nmbAssets, initBudget);
for (t = 1; t < nmbStages-1; t++)
  coreNodes[t] = new MidStageNode(coreNodes[t-1], retData[scenNodeNmb[t]-1]);

coreNodes[t] = new LastStageNode(coreNodes[t-1], retData[scenNodeNmb[t]-1],
                                 capTarget);

RootNode *ptRoot = dynamic_cast<RootNode *>(coreNodes[0]);

ptRoot->make_objective_function();              // Create the objective func.
mpCoreModel.setObjective(ptRoot->objFunction); // Set the objective
mpCoreModel.attach();                           // Attach the model
```

**Fig. 10:** The `main` function, part 1—creating of the core-model object. Note that we have removed declaration of some entities, for the sake of brevity.

After some definitions, we get the node numbers of the nodes in the first scenario, using the `firstLeaf` and `get_parent` methods of the `BinTreeStruct` object `scTree`. Then we create a vector `coreNodes` of $T$ stage-node objects and initiate them with the data from the first scenario. Note how the different derived classes have different parameters: compared to the `MidStageNode` objects, the `RootNode` object does not have any scenario data, while the `LastStageNode` class needs in addition the value of capital target $v$.

Finally we create the objective function by calling the `make_objective_function()` method on the root node and attaching it to the model object `coreModel`. The `attach()` method tells FLOPC++ to process the model and create its OSI representation.

The next step is to create an SMI `SmiCoreData` object with the created core (deterministic) model—see Figure 11 for the source code. To do this, we have to get the number of variables (columns) and constraints (rows) in the model, as well as the stage number of each column and row. This is done using the `all_variables` and `all_constraints` objects of the `StageNodeBase` class; without them, we would have to refer to all the variables and constraints by name.

The bracket operator, used in the last line of the code, is overloaded in the `MP_Model` class to point to the OSI interface to the attached LP model. In other words, the constructor of the `SmiCoreData` class will get an OSI interface to the core model, the number of stages, and the vectors of stage information for all variables and constraints as its parameters.

```
// Get sizes of the created core problem
int nmbCoreCols = mpCoreModel->getNumCols();
int nmbCoreRows = mpCoreModel->getNumRows();

// Get the stage number for all variables and constraints
int *colStages = new int[nmbCoreCols];
for (t = 0; t < nmbStages; t++) {
  for (j = 0; j < (int) coreNodes[t]->all_variables.size(); j++) {
    int colIndx = coreNodes[t]->all_variables[j]->getColumn();
    colStages[colIndx] = t;
  }
}
// The same for constraints
int *rowStages = new int[nmbCoreRows];
for (t = 0; t < nmbStages; t++) {
  for (i = 0; i < (int) coreNodes[t]->all_constraints.size(); i++) {
    int rowIndx = *coreNodes[t]->all_constraints[i];
    rowStages[rowIndx] = t;
  }
}

// Build the core problem as an SMI object
SmiCoreData smiCoreData(mpCoreModel.operator->(), nmbStages,
                        colStages, rowStages);
```

**Fig. 11:** The `main` function, part 2 – creating an SMI core-data object.

Once we have the core model, we are ready to create the stochastic model, using the code presented in Figure 12. The method `get_next_scenario()` provides a pointer to the node indices into the `retData` array for a given scenario, and also provides the parent scenario index, the branching stage, and the probability. These calculations are specific to the particular tree (in our case we are using the method from `BinTreeStruct`). The scenario is then added to the SMI `SmiScnModel` object `stochModel` using the `generateScenario()` method, whose parameters are the core model, the matrix of differences wrt. the parent scenario, branching stage, parent scenario, and the scenario probability.

By the end of the code in Figure 12, the SMI object `stochModel` includes the full stochastic model and is thus ready to be solved. Unfortunately, there is currently no stochastic solver that we could use, so the only option is to create an OSI object of the deterministic equivalent using the `loadOsiSolverData()` method of the `SmiScnModel` class and then solve the problem using one of the OSI-enabled solvers.

Despite this temporary shortcoming, we can still take advantage of the fact that the `SmiScnModel` holds the information about the structure of the stochastic model. We can, for example, compute the objective value of the stochastic program by summing up the scenario objective values, as shown in Figure 13.

Finally, the example in Figure 14 shows the combined use of the SMI object and the FLOPC++ `stageNode` objects: it traverses the whole scenario tree and reports for each scenario the wealth in all the nodes along the scenario. Note how we have to provide, for each node's `get_wealth()` method, the solution vector of the corresponding scenario problem, for the node's model to be able to compute the wealth.

```
SmiScnModel smiModel;  // SMI model object
int *scenNodeNmb;      // scenario NodeNumbers for accessing returns array
int scen;              // scenario number
int parent;            // parent scenario that current scenario branches from
int branch;            // branching stage
double prob;           // probability

while(scenNodeNmb = binTree.get_next_scenario(&scen,&parent,&branch,&prob)) {
  // clean the matrix of differences - must reset dimensions!
  ADiff.clear();
  ADiff.setDimensions(nmbCoreRows, nmbCoreCols);

  for (t=branchStage; t<nmbStages; t++) {
    // load modified data into ADiff
    coreNodes[t]->load_modified_matrix(ADiff,retData[scenNodeNmb[t]-1]);
  }
  //generate scenario -- only the matrix is changed in this application.
  smiModel.generateScenario(&smiCoreData, &ADiff, NULL, NULL, NULL, NULL, NULL,
                            branch, parent, prob);
}
```

**Fig. 12:** The `main` function, part 3 – creating an SMI stochastic object.

```
  double objValue = 0.0;

  for (SmiScenarioIndex sc = 0; sc < smiModel.getNumScenarios(); sc ++) {
    SmiScnNode *smiNode = smiModel.getLeafNode(sc);      // leaf node of scen. sc
    double scProb = smiNode->getModelProb();             // probability of the leaf
    double scenObjVal = smiModel.getObjectiveValue(sc); // objective value
    objValue += scProb * scenObjVal;
  }
```

**Fig. 13:** The `main` function, part 4 – computing the objective function by summing up the scenario objectives.

```
// Report the wealth at each node of the tree
vector<double> nodeWealth(nmbStages, 0);

for (SmiScenarioIndex sc = 0; sc < smiModel.getNumScenarios(); sc ++) {
  // Get the solution for scenario sc sorted into the original (FlopC++) order:
  int nmbColsInScen;
  double *smiScenSol = smiModel.getColSolution(sc, &nmbColsInScen);

  // Go up the tree from leaf node to root
  int nodeStage = nmbStages;
  SmiScnNode *smiNode = smiModel.getLeafNode(sc);
  while (smiNode != NULL) {
    // Get the wealth from the scenario solution
    nodeWealth[nodeStage-1]
      = coreNodes[nodeStage-1]->get_wealth(smiScenSol, nmbColsInScen);
    smiNode = smiNode->get_parent();
    nodeStage--;
  }
  free(smiScenSol);
}
```

**Fig. 14:** The `main` function, part 5 – computing the wealth development along all scenarios.

## 5.6 Final comments

The example shows a basic pattern for combining FLOPC++ and SMI. These are based on the following abstractions:

1. A `StageNode` class that associates constraints and variables with a stage descriptor, and supplies methods for generating scenario data from stochastics data.
2. A `ScenTreeStruct` class that provides stochastics data for consumption by the `StageNode` classes, and provides branching information for SMI.

Readers who are not experienced with C++ will probably be asking themselves at this point whether the example represents any kind of progress towards a modelling environment! It is true that the example contains a lot of C++ coding, so this is a natural question. Let us try to speculate a bit on how the example could evolve towards a modelling system for stochastic programming.

First, we observe that the user should not need to write the `StageNode` classes directly. It is possible to place them into a library and invoke them by FLOPC++ style statements, using an `MP_stage` abstraction:

```
MP_stage T(numStages);
/* .. */
cashFlowBalance(T) =
    wealth(T) == sum(ASSETS, Returns(T,ASSETS) * x(T-1,ASSETS));
```

This abstraction conveys the relationship of constraints and variables to stages. The major piece missing is the mapping between stochastic data and scenario generation, which in our example is implemented by `load_modified_matrix()`. We want a mechanism for the user to specify that the factor `Returns` will be changed when scenarios are generated. One way to do this is to use a label:

```
MP_stochastic Returns(T);
```

With this labelling, the system would know that scenarios will be generated by assigning new values to the factor `Returns`, and would be able to figure out which matrix entries are to be updated. This information could be passed to an object whose function is to take stochastic data as inputs and supply scenario data as outputs.

Second, we note that the main program, which uses SMI methods to generate and solve the stochastic program, is almost completely generic. Most of this code could be placed into a solver class for FLOPC++. The major piece that could not be treated in this fashion is the section that reports the value of the `wealth` variable along each scenario. The difficulty is that FLOPC++ does not have an explicit representation of scenarios. Possibly this could be accomplished abstractly by extending the `display()` method of FLOPC++:

```
for(s = 0; s < MP_scenarios.size(); s++)
for(t = 0; t < T.size(); t++)
   wealth(T).stoch_display(s);
```

The interpretation would be that `MP_scenarios` is an abstract data set that allows access to the underlying `SmiScnModel` object.

Finally, the scenario tree objects could be placed in a library. The `BinTreeStruct` object is perfectly generic, for example. One could supply arguments that describe the up and down movements, as follows:

```
MP_data up(ASSETS);
MP_data dn(ASSETS);
MP_binaryTree Returns(T, ASSETS, up, dn);
```

This would be sufficient to describe a binary tree that has fixed up and down movements. One could even make the up and down movements vary over time.

## 6 Conclusions and future work

In this paper, we have presented a way of modelling stochastic programs in C++, combining the power of SMI and FLOPC++ projects from the COIN-OR open source repository. The stochastic program is modelled using a decomposition into sub-models for nodes of the stochastic tree, plus information about the scenario-tree structure. This allows us to keep track of the structure of the problem and therefore to take advantage of structure-exploiting solvers—once such solvers are available.

While the proposed approach is already applicable, there are several things that can be improved: first of all, we need some structure-exploiting solver to take the advantage of the special structure implied by multiperiod stochastic programs. For this purpose, a Benders' decomposition solver is currently being developed as a part of the SMI project. There is also a plan to interface SMI with the OOPS solver, which is a parallel, structure-exploiting interior point solver, see Gondzio and Grothey (2007). Another way of improving the applicability of the proposed solution is its deeper integration with the SMI and FLOPC++ packages, as mentioned above.

## References

John R. Birge. Stochastic programming computation and applications. *INFORMS Journal on Computing*, 9: 111–133, 1997.

John R. Birge and François Louveaux. *Introduction to stochastic programming*. Springer-Verlag, New York, 1997. ISBN 0-387-98217-5.

J.R. Birge, M.A.H. Dempster, H.I. Gassmann, E.A. Gunn, A.J. King, and S.W. Wallace. A standard input format for multiperiod stochastic programs. *Mathematical Programming Society Committee on Algorithms Newsletter*, 17:1–20, 1987.

C.S. Buchanan, K.I.M. McKinnon, and G.K. Skondras. The recursive definition of stochastic linear programming problems within an algebraic modeling language. *Annals of Operations Research*, 104:15–32, 2001.

C. Condevaux-Lanloy, E. Fragniere, and A.J. King. Sisp: Simplified Interface for Stochastic Programming. *Optimization Methods and Software*, 17:423–443, 2002.

Robert Fourer and Leo Lopes. StAMPL: A filtration-oriented modeling tool for multistage stochastic recourse problems. *INFORMS Journal on Computing*, to appear, 2008. doi: 10.1287/ijoc.1080.0289.

H.I. Gassmann and A.M. Ireland. Scenario formulation in an algebraic modelling language. *Annals of Operations Research*, 59:45–75, 1995.

H.I. Gassmann and E. Schweizer. Proposed extensions to the smps input format for stochastic linear programs. Working paper wp–96–1, Dalhousie School of Business Administration, 1996.

J. Gondzio and A. Grothey. Parallel interior point solver for structured quadratic programs: Application to financial planning problems. *Annals of Operations Research*, 152(1):319–339, 2007.

Tim Helge Hultberg. FLOPC++ An algebraic modeling language embedded in C++. In Karl-Heinz Waldmann and Ulrike M. Stocker, editors, *Operations Research Proceedings 2006*, volume 2006 of *Operations Research Proceedings*, pages 187–190. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-69994-1 (print), 978-3-540-69995-8 (online). doi: 10.1007/978-3-540-69995-8_31.

Peter Kall and János Mayer. Building and solving stochastic linear programming models with SLP-IOR. In S. W. Wallace and W. T. Ziemba, editors, *Applications of Stochastic Programming*, volume 5 of *MPS/SIAM Ser. Optim.*, pages 79–93. SIAM, Philadelphia, PA, 2005.

Bjarni Kristjansson. *MPL Manual*. Maximal Software, `http://www.maximal-software.com`.

Robin Lougee-Heimer. The Common Optimization INterface for Operations Research. *IBM Journal of Research and Development*, 47(1):57–66, 2003. doi: 10.1147/rd.471.0057.

Christian Valente, Gautam Mitra, Mustapha Sadki, and Robert Fourer. Extending algebraic modelling languages for stochastic programming. *INFORMS Journal on Computing*, to appear, 2008. doi: 10.1287/ijoc.1080.0282.

P. Valente, G. Mitra, and C. A. Poojari. A stochastic programming integrated environment (SPInE). In S. W. Wallace and W. T. Ziemba, editors, *Applications of Stochastic Programming*, volume 5 of *MPS/SIAM Ser. Optim.*, pages 115–136. SIAM, Philadelphia, PA, 2005.